A Reversed Reversi Agent based on Genetic Algorithm and MCTS

Shengli Zhou (12212232), CSE, SUSTech

***Abstract*—In this paper, we provide a method of utilizing genetic algorithm and Monte Carlo Tree Search (MCTS) to design an agent that can play the chess game of Reversed Reversi. The algorithm applies genetic-algorithm-trained heuristic function on Minimax Search at the beginning of the game and determines actions by MCTS at the end of the game. Through this method, the two algorithms can complement each other's advantages, thereby improving overall performance.**

***Index Terms*—Artificial Intelligence, Search Algorithm, Genetic Algorithm, Minimax Search, Monte Carlo Tree Search (MCTS).**

I. Introduction

THE origin of Reversi, also known as Othello, dates back to the late 19th century, invented by the British, and later popularized and renamed in the 1970s by a Japanese individual, drawing inspiration from Shakespeare's play "Othello," which reflects the struggle between black and white. [1] Reversed Reversi is a chess game derived from Reversi which is played based on the following rules.

The game starts with the following situation:

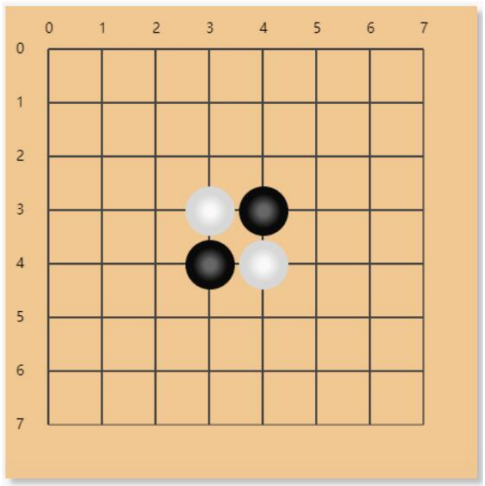

Fig. 1. Initial Board Configuration.

During a game, the black side plays first and the opponents make moves alternatively. During each move, the player puts a chess of his / her color on a valid cross point on the board. A point is considered valid if there is a chess that have the same color on the 8 directions with all cross points between them being occupied by opponent's chess. After the move, all the opponent's chess mentioned above are turned to the player's color. The game ends when the board is full or neither of the players can make a move. The goal of Reversed Reversi is opposite to that of Reversi, i.e., the player with less chess on the board when the game ends wins the game.

Not only is Reverse Othello a chess game, but it also expresses the idea that merely relying on brute force to solve problems with a single goal in mind may not yield satisfactory results. On the contrary, approaching each step strategically may lead to better outcomes.

In this report, we propose a method for designing an agent that can play Reversed Reversi based on Minimax Search with genetic-algorithm-trained heuristic function and Monte Carlo Tree Search (MCTS). During the first phase of the game, the search space is large and MCTS does not have good performance as the simulation on the search tree is sparse. Thus, we apply heuristic Minimax Search to overcome this issue. However, when the game is about to end (e.g., after the 40th or 50th play), MCTS can make better choices as the heuristic function may not lead to the optimal decision. Therefore, by combining the algorithms, they can complement other's advantage and maximize the utility.

II. Preliminary

A. Problem Formulation

Though Reversed Reversi can also be played on boards having size other than 8×8, we mainly consider the case of the default size (i.e., 8×8) for simplicity.

To formulate the problem, we regard the board as an 8×8 array B with each element representing a cross point. We denote a black chess as -1 in the array and a while chess as 1 in the array. Unoccupied positions are filled with 0 initially. For example, Table I represents the array corresponding to the initial board configuration.

TABLE I
Initial Board Configuration in Array Format

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	1	-1	0	0	0
0	0	0	-1	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

This project was supported by Department of Computer Science and Engineering, Southern University of Science and Technology (SUSTech). The project was posted in course CS311 Artificial Intelligence (H) as the first course project on March 15, 2024. All works of the project was finished on April 12, 2024.

S. Zhou is a sophomore student in Southern University of Science and Technology (SUSTech), Shenzhen, Guangdong Province, China. (e-mail: zhousl2022@mail.sustech.edu.cn).

During a move operated by player x (x is 1 or -1), the player change a 0 in the array to x. $B[x_0, y_0]$ can be changed if there exist $(\Delta x, \Delta y) \in \{(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1)\}$ and positive integer h s.t. $(B[x_0 + h\Delta x, y_0 + h\Delta y] = x) \wedge (\forall_{i\in[1,h)} B[x_0 + i\Delta x,\ y_0 + i\Delta y] = -x)$. Once the player decides to perform a move $B[x_0, y_0] := x$, then $B[x_0 + i\Delta x, y_0 + i\Delta y] := x$ will be performed automatically for all $i \in [1, h)$.

A player can skip a move only if he / she cannot make any move. When both players cannot make a move, the game ends. When a game ends, black wins if the sum of all elements on the board is greater than 0, white wins if the sum of all elements on the board is smaller than 0. Otherwise, the game ends with a tie.

B. Definition of Terms and Related Algorithms

Stable Chess We define a chess piece as a stable chess if it cannot be flipped (i.e., change to the other color) no matter what moves are done in the subsequent steps.

Consider if there is a chess in $B[0,0]$ (denote $x := B[0,0]$), then the chess is trivially a stable chess. Under this circumstance, let y_0 be the smallest number s.t. $B[0, y_0] \neq x$, then $B[0, y]$ for all $0 \leq y < y_0$ are all stable chess. For the second row, the situation is similar, except that the horizontal ordinate cannot be greater than $y_0 - 2$ (here y_0 is the above-mentioned value calculated in the first row). The additional constraint is due to the case that the opponent's chess in $B[0, y_0]$ may flip the chess in $B[1, y_0 - 1]$. By applying this method to each row (and also symmetrically to each column), we can get the following algorithm for calculating stable chess:

Algorithm 1 Calculate Stable Chess

```
x ← B[0,0]
if x == 0 then
    return ∅
end if
y_0 ← 8
for i = 0 to 7 do                          ▷ Iterate through rows
    y_1 ← smallest y_1 s.t. B[i, y_1] ≠ x
    y_0 = min{y_0 − 1, y_1}
    Mark B[i, 0] ∼ B[i, y_0 − 1] as stable chess
end for
x_0 ← 8
for j = 0 to 7 do                          ▷ Iterate through columns
    x_1 ← smallest x_1 s.t. B[x_1, j] ≠ x
    x_0 = min{x_0 − 1, x_1}
    Mark B[0, j] ∼ B[x_0 − 1, j] as stable chess
end for
return Set of stable chess
```

From the analysis above, we can come to the conclusion that all stable chess are related to an occupied corner. Thus, we can use similar method to count the stable chess that are related to the other three corners.

Generally speaking, having too much stable chess on the board is not a wise choice in Reversed Reversi, since the player wants the opponent to flip his / her chess and decrease the number of his / her chess.

Frontier Chess We define a chess as a frontier if there is at least one blank position in its 8-adjacent positions, i.e., if $\exists(\Delta x, \Delta y) \in \{(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1)\}$ s.t. $B[x+\Delta x, y+\Delta y] = 0$ for some $B[x, y] \neq 0$, then $B[x, y]$ is a frontier chess. As the number of a player's frontier chess increases, it is more likely that his / her chess will be flipped, thus reducing the number of his / her chess on the board.

Mobility We define mobility as the number of choices a player have under a certain situation. When a player's mobility is small, it is generally a disadvantage to him / her since the remained positions are usually designed to be choices of low utility.

III. Methodology

A. General Workflow

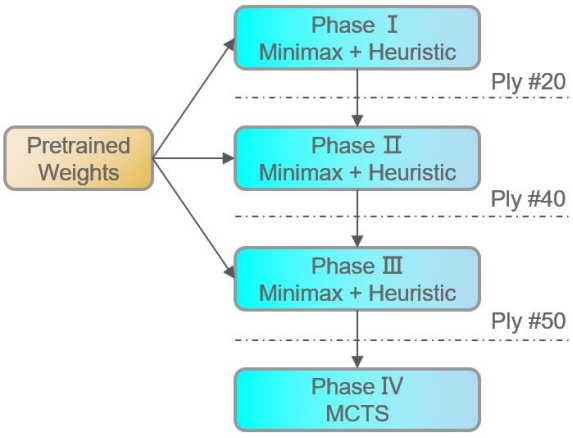

Fig. 2. General Workflow.

The proposed model divides the game into four phases. During the first three phases (from the first play to the 50th play), the model use Minimax Search with $\alpha - \beta$ pruning and heuristic function to make decision. In the last phase (from the 51st play to the 60th play), the model use MCTS to find the solution. Also since the strategy varies a lot during the game, we further divide the first 50 plays as three phases and use Genetic Algorithm to train a set of weights for each of the phases.

B. Detailed Algorithm Design

1) Weight Design: The three sets of weights mentioned above contains 20 parameters in total.

Positional weight (10 parameters, shared) Different positions on the board have different potential utilities, e.g., chess piece on the boundary of the board is likely to become a stable chess (while they can also be the opponent's stable chess if being flipped). Thus, we train a set of parameters to evaluate the utility of each position. In order to decrease the number epochs needed in training, we assign the same weight to some different positions to reduce the number of parameters

based on rotational symmetry and axial symmetry. Denote the parameters as $a_0 \sim a_9$, then the detailed configuration is shown in Table II below.

TABLE II
POSITIONAL WEIGHT CONFIGURATION

a_9	a_8	a_6	a_3	a_3	a_6	a_8	a_9
a_8	a_7	a_5	a_2	a_2	a_5	a_7	a_8
a_6	a_5	a_4	a_1	a_1	a_4	a_5	a_6
a_3	a_2	a_1	a_0	a_0	a_1	a_2	a_3
a_3	a_2	a_1	a_0	a_0	a_1	a_2	a_3
a_6	a_5	a_4	a_1	a_1	a_4	a_5	a_6
a_8	a_7	a_5	a_2	a_2	a_5	a_7	a_8
a_9	a_8	a_6	a_3	a_3	a_6	a_8	a_9

Let the matrix presented in Table II be P. During evaluation, we use the sum of elements in the result of element-wise multiplication between P and B times the number representing opponent's color to estimate the utility w.r.t. positional factors, i.e. positional score = (opponent's color) $\times \sum_{i=0}^{7}\sum_{j=0}^{7} P_{i,j}B_{i,j}$.

Stability weight (1 parameter, shared) Since positions where stable chess pieces lie are excluded from the scope of consideration, only the difference between the number of stable chess of the players matters. Therefore, we use stability weight $\times$ (number of opponent's stable chess - number of own stable chess) to represent the utility w.r.t. stable chess. We define this value as stability score.

Mobility weight (2 parameters for each phase, 6 parameters in total) Let M_1, M_2 be the two mobility weight in a phase of the game, we use mobility score = $M_1\times$ (own mobility) $-M_2\times$ (opponent's mobility) to evaluate the utility w.r.t. mobility.

Frontier weight (1 parameter for each phase, 3 parameters in total) Here we use frontier score = (frontier weight) $\times$ (opponent's frontier - own frontier) to evaluate the utility w.r.t. the number of frontier chess.

Finally, we use the sum of all scores to evaluate the utility of a state.

2) Weight Pretraining: In our method, we use Genetic Algorithm to pretrain weights for evaluation.

During each training epoch (i.e., generation for evolution), we use 64 agents with different parameters (i.e., genes) and play a round robin solely using the strategy of Heuristic Minimax Search (which will be mentioned in the next section), i.e., each two different agents will play 2 games on different sides. If an agent wins a game, then the score of the agent will be increased by 1, if an agent loses a game, then the score of the agent will be decreased by 1, otherwise the score will not change. The score for an agent is the sum of all scores in the games it plays.

After round robin, agents will perform crossover operation to generate offspring. When two agents perform crossover, their offspring will randomly choose each parameter from one of its parents. After that, there is also small probability that the offspring will perform mutation.

When selecting the agents to remain in the next epoch, we adopt elitist preservation strategy to keep the agents with high scores from the last epoch. Though, theoretically, elitist preservation strategy can ensure the performance of the agents become better monotonically as the number of training epoch increases, we claim that the performance of agent may not keep becoming better when competing with other agents on the Online Battle Platform. This is because the score of an agent is based solely on its performance when competing with other agents in the same epoch, so agents with the highest score is likely to be good but may not be optimal. Furthermore, as other agents on the Online Battle Platform have different strategies, agents using parameters from later epochs may not perform better than those using parameters from former epochs.

In our method, we trained the agents for 3000 epochs and choose the set of parameters via the performance of agent on the Online Battle Platform.

Algorithm 2 Genetic Algorithm

```
agents ← generate 64 agents
for 3000 epochs do
    for { agent1, agent2 } ⊆ agents do
        winner, loser ← play(agent1, agent2)
        score[winner] += 1
        score[loser] -= 1
    end for
    save scores in a checkpoint file
    agents ← filter(agents, crossover(agents))
end for
```

3) Minimax Search: During the first three phases of the game, we use Minimax Search with $\alpha - \beta$ pruning to make the decision. Before each move, the algorithm search for four steps in the search space in advance. When reaching in the fourth step, we will use a heuristic function characterized by the pretrained weights to evaluate the utility.

During the pretraining process, as we solely use Heuristic Minimax Search as our strategy, we also need to handle the situation where a game ends. Here we use 10000 and -10000 as the utility for winning and losing a game respectively and 5000 as the utility for making a tie. These hyper-parameters are chosen based on two constraints: large hyper-parameters may cause the weights to reach the upper-bound too easily while small hyper-parameters will make weights hard to train due to accuracy issues. Thus, we choose the above-mentioned hyper-parameters to avoid these two problems.

4) MCTS: In the last phase of the game, we use MCTS to make the decision. During each step, we kept running simulation until 95% of the time limit (i.e., 4.75 seconds for default) has been used. Here we choose $\sqrt{2}$ as our coefficient in calculating Upper Confidence Bound (UCB), i.e.,

$$\text{UCB} = \frac{U_i}{N_i} + \sqrt{\frac{2\log N}{N_i}} \tag{1}$$

where U_i is the number of winning games after reaching this state during simulation, N_i is the number of simulations that have enter this state and N denotes the total number of simulations.

C. Analysis

Minimax Search is a conservative algorithm that assumes the opponent is smart enough to make each step optimal.Minimax Search can maximize the lower-bound of utility but may not use the strategy with the highest expected utility (i.e., the highest probability of winning) to avoid risks. On the contrary, MCTS is designed to choose the strategy that has the highest probability of winning.

When agents use Minimax Search to make decisions, since the search space is too large, using a heuristic function to evaluate the utility of intermediate statuses is inevitable. Also since different agents have different heuristic functions, the assumed worst-case scenario in Minimax search often does not occur, which may result in the algorithm missing some chances of winning.

However, MCTS also has its shortcomings, especially in the early stages of the game. Through theoretical analysis and experiments on the Online Battle Platform, we conclude two reasons for the unsatisfactory in the performance of MCTS.

- Though we try to maximize the number of simulations, it is still insufficient for estimating the probability of winning since the simulations are sparse when we look at deeper levels on the Monte Carlo Search Tree.
- The first two phases in the game is vital to the improving the probability of winning, which is also strategical. But since MCTS is uninformed of the goal, it may make moves that can be easily overturned by the opponent.

Thus, an optimal combination of Minimax Search and MCTS is applying Minimax Search in the former phases (i.e., the first three phases) of the game while using MCTS in the latter phase (i.e., the fourth phase). After applying all these strategies, we only need to find an optimal division between the last two phases, which we will discuss in the next section.

IV. Experiments

During this part, we conduct an experiment to determine the optimal division between the last two phases.

A. Experimental Setup

1) Environment: The code for performing this experiment is written in Python 3.8.18 and can be executed under an environment with numpy==1.24.4.

2) Fixed Parameters: According to the principle of single variable, we use the same parameters for the heuristic functions of Minimax Search. The parameters are shown below:

- Positional weight: 7.679, -1.842, 25.927, -8.150, 16.158, 50.0, 37.592, 13.971, -50.0, 50.0.
- Stability weight: 120.0.
- Mobility weight: 3.910, 6.881, 1.824, 21.950, 7.710, 50.0.
- Frontier weight: -0.934, 2.896, 21.691.

We choose this set of parameters since they have relative good performance on the Online Battle Platform.

TABLE III
Result of Round Robin

No.	CP1	CP2	CP1 wins	CP2 wins
1	60	50	20	20
2	60	40	20	20
3	60	30	24.5	15.5
4	60	20	34	6
5	60	10	33	7
6	60	0	35	5
7	50	40	31	9
8	50	30	39	1
9	50	20	38	2
10	50	10	39	1
11	50	0	39	1
12	40	30	28	12
13	40	20	36	4
14	40	10	36.5	3.5
15	40	0	37	3
16	30	20	28	12
17	30	10	24	16
18	30	0	31	9
19	20	10	25	15
20	20	0	21	19
21	10	0	24	16

TABLE IV
Statistics of each Agent

CP	Win as Black	Win as White	Total
0	25	28	53
10	35.5	31	66.5
20	35	35	70
30	52.5	59	111.5
40	90	76.5	166.5
50	118	88	206
60	105	61.5	166.5

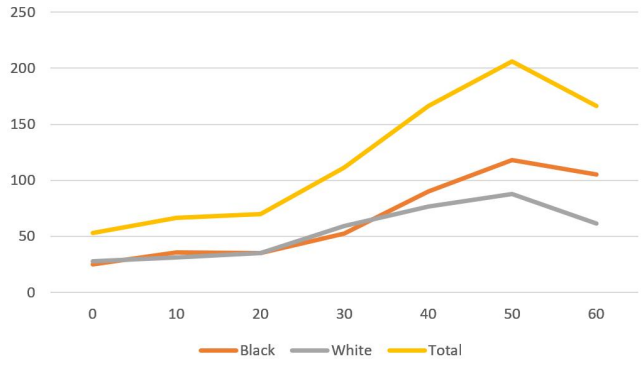

Fig. 3. Number of winning games for each agent.

B. Results

We denote conversion point between phase III and phase IV as CP. We generate 7 agents with CP equal to $0, 10, 20, 30, 40, 50, 60$ respectively. Each two different agents i, j will play 40 games, including 20 games with i as black and j as white in 20 games played in the opposite configuration. After all games are played, we count the number of winning games for each agent to determine the optimal conversion point.

The result of the experiment is shown in Table III and Table IV above. When a tie happens, we regard each player wins 0.5 game.

From the result of Experiment 4 to Experiment 18 shown in Table III, agents with CP equal to $40, 30, 20, 10$ and 0 are significantly worse then agents with CP equal to 60 and 50. Though agents with CP equal to 60 and 50 have similar performance during direct confrontation, the agent with CP equals to 50 have better performance considering the overall winning rate according to Table IV. Therefore, we choose CP $= 50$ as the hyper-parameter.

C. Analysis

According to Figure 3, as CP of the agent increases, the performance first increases and then decreases, reaching its maximum when CP $= 50$. The trend of performance increase and decrease is consistent with the predicted outcome discussed in the III-C part. From theoretical analysis and results of experiments, we can make the following conclusions:

- For agents that starts using MCTS before the 40th play, the search space is too large for MCTS to reach sufficient simulation density for making a decision of high utility.
- The first few phases of the game is strategic where simply pursuing high utility may cause the agent to make decision that can easily be overturned by the opponent. Under this circumstance, the conservative Minimax Search can do better than MCTS.
- During the last phase of the game, since MCTS can make enough simulations for the agent to make a good decision while Heuristic Minimax Search still cannot ensure the optimality of decisions, the performance of MCTS can exceed that of Minimax Search. Thus, changing the algorithm from Minimax Search to MCTS in the last 10 plays can maximize the overall performance of the agent.

V. Conclusion

In this paper, we design an agent for playing Reversed Reversi based on Minimax Search and MCTS, we also utilize Genetic Algorithm to train the parameters for the Heuristic function of Minimax Search. We also discuss the advantages and shortcomings of Minimax Search and MCTS and use experiments to determine the best strategy of combining Minimax Search and MCTS. The results of the experiments match our predictions and prove that our analysis is correct.

Though the code of this project is required to be uploaded with a single .py file, but organizing the codes in the form of a project can greatly improve the clarity of the structure and contribute to higher efficiency when coding. Moreover, fully utilizing packages in Python instead of implementing basic modules by ourselves can make the code run faster as there are a great deal of optimization embedded in the implementation of Python.

Apart from Genetic Algorithm, Minimax and MCTS, there are still several ways to improve the performance of the agent, including introducing chess manual, utilizing Self-Adversarial Training and using neural network to extract chessboard's feature. We hope to try these methods in the future and compare them with our current model.

At last, there is still an inexplicable issue, some sets of parameters trained by Genetic algorithm performs better when on the black side, while some sets of parameters are better when playing as white. In the future, we will also try to explain this phenomenon by the mechanism of Genetic Algorithm.

Acknowledgments

The code for training weights via Genetic Algorithm is partially adapted from https://github.com/Tokasumi/CS303-Fall2021-Reversed-Reversi.

I would like to thank Professor Bo Yuan for his informative lectures and guidance through the project.

I would also like to thank laboratory teacher Yao Zhao for the inspiring suggestions to this project.

In addition, I would like to thank Teaching Assistants Xiang Yi, Tingjing Zhang and Yibo Tang for the tutorial sessions and patient question-answering.

References

[1] *ERNIE Bot.* (March 2024). Baidu. Accessed: Mar. 30, 2024. [Online]. Available: https://yiyan.baidu.com

[2] *Tokasumi, "Tokasumi/CS303-Fall2021-Reversed-Reversi," GitHub.* (Mar. 27, 2024). Accessed: Apr. 2, 2024. [Online]. Available: https://github.com/Tokasumi/CS303-Fall2021-Reversed-Reversi